JOSHUA COOK

LATENT SEMANTIC ANALYSIS

Contents

From a product perspective, we seek a command line interface (CLI) that can be used to query a dataset consisting of documents. Our underlying dataset against which we will be making queries are the pages comprising Wikipedia. A user will be able to use our command line tool to receive the n pages in a particular subset of Wikipedia pages that most closely match a passed serch term.

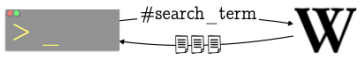
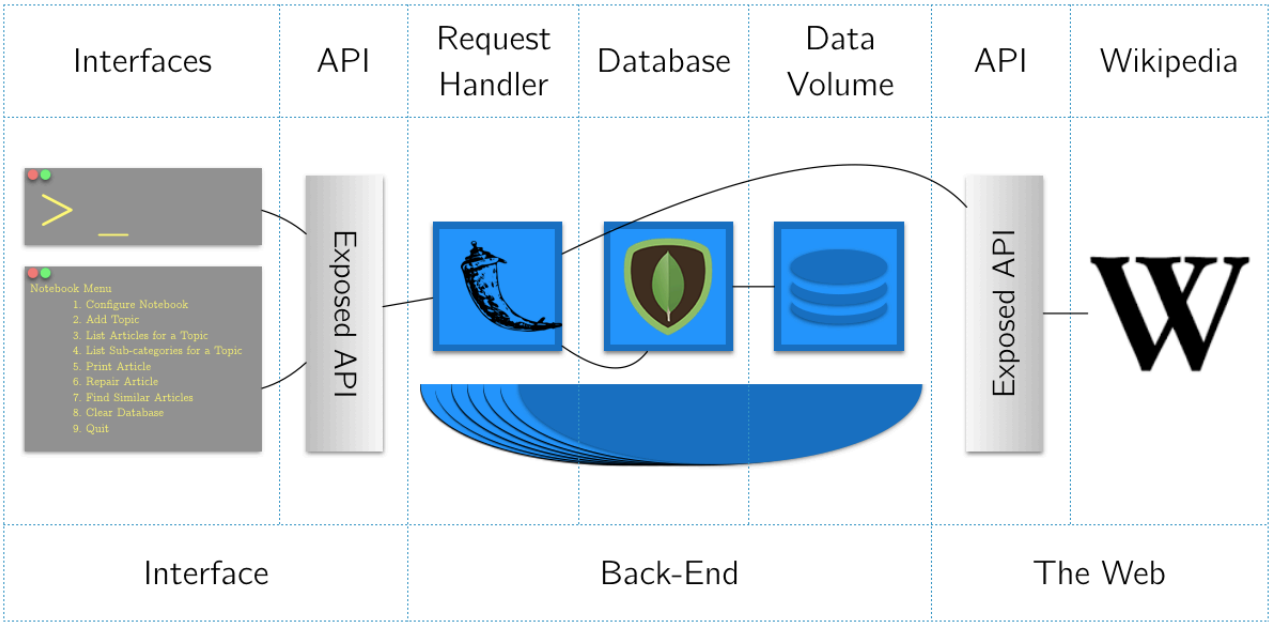


Figure 1: A rough sketch of our system

System Design

A Minimal System



We can think of the components required to build our system as falling into one of two categories – Interface Components and Back-End Components.

Interface Components Users will use one of three different systems to interface with the deeper Back-End components.

1) a *Command Line Interface* which will take basic commands such as `download`, `display`, and `search` and return results directly to the shell. Output could then be piped to files or other processes.
2) a command line UI or *Notebook* where users can display entities existing in the system - Categories, Pages, or Queries. The notebook is available via the command line argument `notebook`
3) an exposed API through which the system can be manipulated. Strictly speaking, both the CLI and the Notebook are using the API to perform system actions. That said, the system could also be directly manipulated via API calls from `curl`, Postman, or even a web-page.

Back-End Components For portability, we have decided to use microservice containerization via the Docker toolset to design our system. We designed the system using `docker` and `docker-compose` and designed our CLI tools to leverage these technologies as well. End users will need to install the Docker toolset on their local machine in order to use the system, but having done so will no longer need to be concerned with regard to system dependencies.

Docker is a system for developing applications such that the local development environment, as well as any possible environment into which we would deploy the application, are identical. Docker solves this problem via virtualization. We can not guarantee that my development machine will be running the same OS as my deployment machine, but I can guarantee that they can both run the Docker daemon.

Our system has three core components, each of which is the sole process running in a Docker Container.

1) a *Request Handler*

   We will be building this image ourselves on top of the public Python 3.5 image.

2) a *Database* into which we will store the objects with which we are interacting.

   We will be using the latest public MongoDB image.

3) a *Data Volume* which will serve no other purpose than to act as storage for our Database

   We will be using the public image `cogniteev/echo`.[1]

[1] While the author wrote this about an earlier project `cogniteev/true`, the purpose of this image can be inferred from this post: http://dockermeetupsinbordeaux.github.io/docker-compose/data-container/2015/03/01/minimalistic-docker-data-container.html

We intend to run this system using the Docker daemon using the command line tool `docker-compose`. With this tool, designing the system is as simple as the following `docker-compose.yml`.

```
notebook:
 image: joshuacook/miniconda
 volumes:
   - .:/source
 links:
   - mongo
mongo:
 image: mongo
 volumes_from:
   - mongodata
mongodata:
 image: cogniteev/echo
 command: echo 'Data Container for MongoDB'
 volumes:
   - /data/db
```

We run the system using this command from our project directory

```
$ docker-compose up
```

A Single `download` Transaction

Let us follow a single `download` transaction making its way through our system.

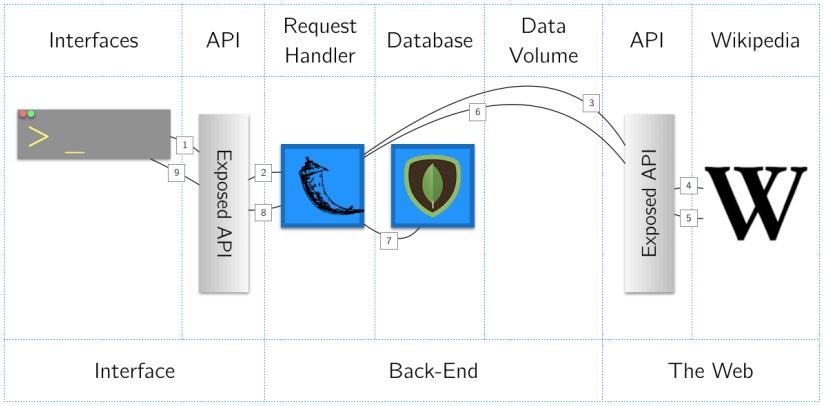


Figure 2: A Single `download` Transaction

1. a User issues a `download` request with a `category` and optionally a `depth` via the CLI
2. the CLI makes a RESTful request containing the parameters of the Request Handler
3. the Request Handler receives the `download` request and makes a RESTful request of Wikipedia's API using the Category provided
4. Wikipedia receives the request and using the Category requests page information from the Wikipedia server
5. the Wikipedia server processes the request and returns page information to the request
6. the Request Handler receives the requested data from Wikipedia
7. the Request Handler writes the received data to the Database
8. the Request Handler responds to the request made by the CLI
9. the CLI is updated with the response from the Request Handler

A Single `search` Transaction

Let us follow a single `search` transaction making its way through our system.

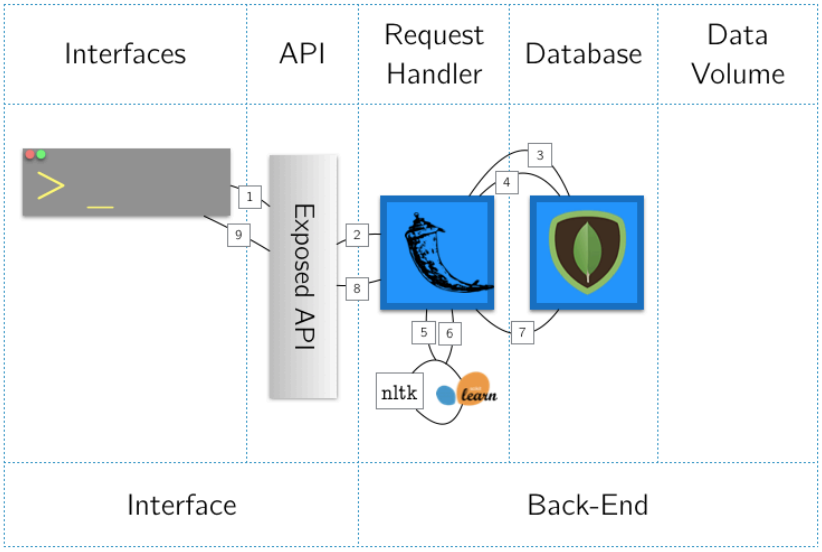


Figure 3: A Single `download` Transaction

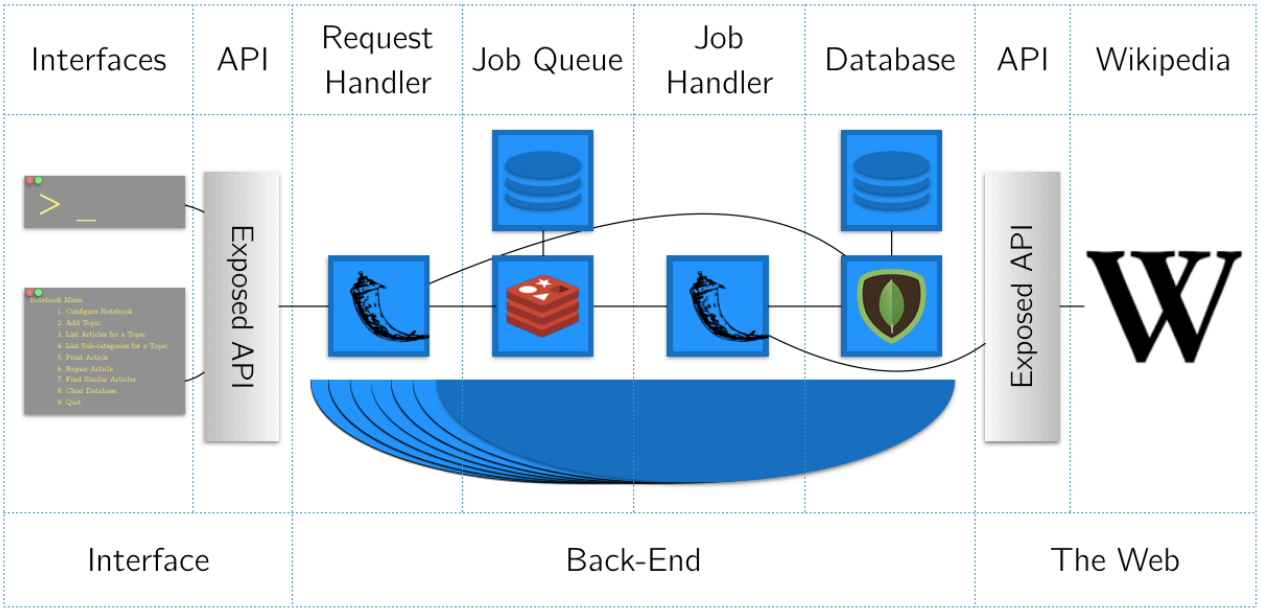
1. a User issues a `search` request via the CLI containing a `category`, a `search_term`, and optionally a `depth`.
2. the CLI makes a RESTful request containing the parameters of the Request Handler
3. the Request Handler receives the `search` request and queries the Database for the appropriate set of pages
4. the Database returns the pages requested
5. the Request Handler passes the pages requested to `nltk` and `scikit-learn` for processing
6. `nltk` and `scikit-learn` return the query results to the Request Handler
7. the Request Handler writes the results of the query to the Database
8. the Request Handler responds to the request made by the CLI
9. the CLI is updated with the results of the query from the Request Handler

Additional Vertical Scaling

Such a system is not particularly well scalable. Though it is not built in the accompanying repository of code. We recommend adding three additional Back-End components:

1. A *Job Handler*
2. A *Job Queue*
3. A *Data Volume* for the Job Queue for a scaling of our system

In this model, the Request Handler is responsible for passing delayed jobs to the Job Queue. The Job Handler will receive then execute jobs. The original request will be completed when the job is passed to the Job Queue. An additional mechanism for querying job status is added to the CLI.



Development process

Building the Request Handler Image in Docker

Docker development arises from the tension between developing an application locally and deploying it remotely. Modern programming languages have developed solutions to this issue in techniques for describing an application's requirement ecosystem. Ruby uses `rvm` or `rbenv`. Python uses `virtualenv` and `requirements.txt`. With Docker none of these are necessary. Instead we use a simple DSL to describe the environment in which our application will run, then use that environment for all stages of our applications' lifetime – development, testing, deployment.

The process for designing our Docker environment is fairly straightforward:

1. write a `Dockerfile`
2. build that Docker image using the `docker build` command

```
$ docker build -t joshuacook/miniconda .
```

If the build is successful, we are ready to use our image. We can launch a new instance of our image using

```
$ docker run joshuacook/miniconda
```

As new dependencies arise in our application, we can simply add them to the end of our `Dockerfile` and rebuild. Because Docker stores our images as a series of layers, one for each command in our `Dockerfile`, the build process will not start from scratch each time and rather will simply run steps that are different from our previous iteration.

THE JOSHUACOOK/MINICONDA DOCKERFILE

```
# based on [progrium/busybox](https://github.com/progrium/busybox)
# and [Anaconda Python Distribution](https://store.continuum.io/cshop/anaconda/)

FROM progrium/busybox
MAINTAINER @joshuacook
RUN opkg-install bash bzip2
ADD conda_install.sh /root/conda_install.sh
RUN ["bash", "/root/conda_install.sh"]
ENV PATH /root/miniconda3/bin:$PATH

# http://bugs.python.org/issue19846
ENV LANG C.UTF-8
ENV PYTHONIOENCODING utf8

CMD [ "/bin/bash" ]

RUN conda install --yes \
    'pandas=0.17*' \
    'scipy=0.17*' \
    'scikit-learn=0.17*' \
    'cython=0.23*' \
    'nltk=3.2.1'

RUN conda install -c jjhelmus --yes feather-format=0.1.0

RUN conda install beautifulsoup4

RUN conda install ipython

RUN conda install -c blaze mongoengine=0.10.0

RUN mkdir /source
WORKDIR /source
```

Build Image. the minimal image upon which we will run our system. may need to update during development. think of this as replacing virtualenv

Additional Development Tools

In addition to the development tools `docker` and `docker-compose`, we highlight the following tools as central to our development process.

Makefiles

If there is a downside to development in Docker, it is the addition of a layer of abstraction to the running of a development environment. Once the nature of the system has been grokked i.e. the fact that all systems being developed are actually running in a virtual machine, this difficulty is reduced to having to type some fairly lengthy commands in order to execute code. For example, in order to open a shell to your development environment, whereas in the past it may was as simple as opening a terminal application and navigating to the project working directory, now this must be done via Docker. And in fact, the command to do this is fairly long

```
$ docker run -it joshuacook/miniconda /bin/bash
```

This has the effect of launching a new instance of the specified image and running it interactively (`-i`), attaching a pseduo-TTY (`-t`), before launching `/bin/bash` as PID 1. As accessing the development environment is a frequent task, I have added this process to a `Makefile` that sits in the root directory of the projet. Now, attaching a shell to the project is as simple as

```
$ make bash
```

Additionally, I have taken cues from Ruby and its `Rakefile` system[2] and have added mechanisms for other common tasks such as running tests.

[2] Which no doubt took its cues from `Makefiles`

THE MAKEFILE

```
CATEGORY ?= Machine_learning
DOCUMENT_TO_MATCH ?= Universal_portfolio_algorithm
N ?= 10

default: pdf

bash:
    docker run -it -v $(shell pwd):/source --name bash \
        joshuacook/miniconda /bin/bash
    docker rm bash

clean_docker:
    docker rm $(docker ps -aq)

jupyter:
    docker run -v $(shell pwd):/home/jovyan/work \
        --name jupyter -d -p 80:8888 \
        joshuacook/datascience

notebook:
    docker run -it -v $(shell pwd):/source \
        --name miniconda joshuacook/miniconda \
        python src/notebook.py
    docker rm miniconda

pdf:
    pandoc doc/latent_semantic_analysis.md \
    -t latex \
    -H doc/fix.tex \
    --toc \
    -f markdown+tex_math_double_backslash \
    -o doc/latent_semantic_analysis.pdf \
    --latex-engine=xelatex

test_binaries:
    python3 test/binaries.py

watch_and_make:
    while true; do kqwait doc/latent_semantic_analysis.md; make; done
```

Software Infrastructure

the Command Line Interface

the Command Line UI

the Request Handler

Our request handler must perform three essential tasks:

- receive requests and use these to execute a predefined task
- interact with a database storing information about our objects
- interact with the Wikipedia API to retrieve information for Categories and Pages

To address the first, we use a RESTful routing design pattern via the `flask` framework sitting atop the `wsgi` library. To address the second, we use the `mongoengine` framework as a lightweight Object Relational Mapper (ORM) to interact with our Mongo Database. We realize that it is non-traditional to use an ORM to interact with a NoSQL database, but find the having an additional layer of scrutiny on the data going into our system helps prevent issues with the data we will ultimately retrieve. To make requests of Wikipedia we use the library `request` explicitly designed to serve this purpose. Finally, to perform our search on data we have stored in our system we leverage two excellent and very mature libraries for natural language processing and machine learning respectively, `nltk` and `scikit-learn`.

The use of Flask gives us a lot of flexibility in the design of our project. That said, we have take some cues from our work with Ruby on Rails in designing the codebase that will comprise our application.

Our Flask application

```
+-- app
|   +-- controllers
|   |   +-- __init__.py
|   |   +-- application_controller.py
|   |   +-- page_controller.py
|   |   +-- category_controller.py
|   |   +-- query_controller.py
|   +-- helpers
|   |   +-- __init__.py
|   |   +-- application_helper.py
|   |   +-- page_helper.py
|   |   +-- category_helper.py
|   |   +-- query_helper.py
|   +-- models
|   |   +-- __init__.py
|   |   +-- page.py
|   |   +-- category.py
|   |   +-- query.py
|   +-- __init__.py
|   +-- app.py
```

Object models

Our application is principally concerned with three discrete classes of object – `Category`, `Page`, `Query`. Using `mongoengine` will provide sufficient abstraction to interact with MongoDB. We simply need to design the objects we expect as follows.

THE PAGE MODEL, `app/models/page.py`

```
from mongoengine import Document, EmbeddedDocument, \
                        IntField, ListField, StringField

class CategoryTag(EmbeddedDocument):
    category      = StringField()
    depth         = IntField()

class Page(Document):
    pageid        = IntField()
    displaytitle  = StringField()
    text          = StringField()
    url           = StringField()
    category_tags = ListField(EmbeddedDocumentField(CategoryTag))
```

THE CATEGORY MODEL, `app/models/category.py`

```
from mongoengine import Document, IntField, StringField

class Category(Document):
    pageid        = IntField()
    title         = StringField()
    subcategories = ListField(StringField())
```

THE QUERY MODEL, `app/models/query.py`

```
from mongoengine import DateTimeField, Document, IntField, StringField

class Query(Document):
    creation_date      = DateTimeField()
    name               = StringField()
    number_of_matches  = IntField()
    depth              = IntField()
    page_count         = IntField()
    subcategory_count  = IntField()
```

Application Controllers and Routes

Map of RESTful API endpoints

Verb	URI Pattern	Controller#Action
GET	/page.json	page#index
GET	/page/:id.json	page#read
GET	/category.json	category#index
GET	/category/:id.json	category#read
GET	/query.json	query#index
POST	/query.json	query#create
GET	/query/:id.json	query#read
PUT	/query/:id.json	query#update
DELETE	/query/:id.json	query#destroy

page#index

This method will return a JSON object containing stored pages (`title` and `pageid`). Optionally, a number of pages, `n`, a `category`, and/or a `depth` can be specified. If a category is specified with no depth, the depth will default to 0 and return only pages belonging to that category. Will return success code `200 OK` and the JSON object. Parameters must be passed as query parameters e.g.

```
/page.json?n=10&category=lunch&depth=2
```

Optionally, an `offset` can be specified, in which case the given `offset` of `n` pages will be returned e.g.

```
/page.json?n=10&category=lunch&depth=2&offset=2
```

will return categories 21 through 30.

page#read

This method will return a JSON object containing all information for the specified page. Will return success code `200 OK` and the JSON object. No parameters are necessary.

category#index

This method will return a JSON object containing `n` categories. Will return success code `200 OK` and the JSON object. Parameters must be passed as query parameters e.g.

```
/category.json?n=10
```

Optionally, an `offset` can be specified, in which case the given `offset` of `n` sub-categories will be returned e.g.

```
/category.json?n=10&offset=2
```

query#index

This method will return a JSON object containing the last `n` queries executed. Parameters must be passed as query parameters e.g.

```
/query.json?n=10
```

Optionally, an `offset`, `type`, `category`, `search_term` and/or `depth` can be specified for further filtering of results e.g.

```
/query.json?n=10&offset=2&type=download&category=lunch&depth=2
```

or

```
/query.json?n=10&type=search&category=lunch
```

query#create This method will create a new query and is the workhorse of our API endpoints. Two types of queries can be created:

- `download`, used to retrieve categories and pages from Wikipedia
- `search`, used to run a search against pages and categories that have already been downloaded into our system

A successful `download` request will have a defined `category` and `depth`, e.g.

```
/query.json?n=10&type=download&category=lunch&depth=2
```

A successful `search` request will have a defined `category`, `search_term`, `number_of_matches` and `depth`, e.g.

```
/query.json?n=10&type=search&category=lunch&search_term=Free_lunch&number_of_matches=3&depth=1
```

The controller will perform the query, log the results in the database, return success code `201 CREATED` and a JSON object describing

the newly created instance.

`query#read` This method will return a JSON object containing a single `query` instance of that type. The instance is referenced by the trailing `id` in the url. Will return success code `200 OK` and the JSON object.

Application Controllers

https://en.wikipedia.org/w/api.php?action=query&prop=extracts&explaintext&titles=Stack%20Overflow&exlimit=max
https://en.wikipedia.org/w/api.php?action=mobileview&prop=sections&sections=all&page=Stack%20Overflow
Algorithms
Latent Semantic Analysis
K Nearest Neighbors
Sparse Matrices
Inverted Index
Cosine Similarity
Feature Hashing -> XGBoost
TF-IDF Vectorizer
Analysis
Timing of System
Timing of Algorithm
ScalingA